1834 Mid-Year Report

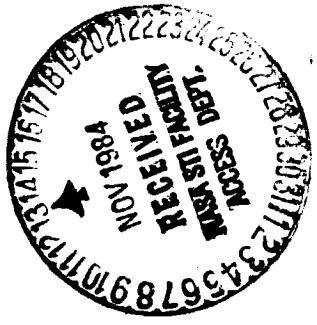
NASA Grant NAG 1-138

SAGA: A Project to Automate the Management of Software Production Systems

Principal Investigator
Roy H. Campbell

Research Assistants
Wayne Badger
Carol S. Beckman
George Beshers
David Hammerslag
John Kimball
Peter A. Kirslis
Hal Render
Paul Richards
Robert Terwilliger

University of Illinois
Department of Computer Science
1304 W. Springfield Ave.
Urbana, IL 61801-2987.
217-333-0215



ABSTRACT

This report details work in progress on the SAGA project during the first half of 1984.

TABLE OF CONTENTS

1. Summary

This report describes the current work in progress for the SAGA project. The highlights of this research are:

- Completion of the prototype SAGA Editor; commencement of testing.
- Completion of the prototype SAGA Symbol Table Manager; commencement of testing.
- Completion of the RRP version of Olorin, the SAGA parser generator; commencement of testing.
- Completion of prototype delta generator for SAGA; commencement of testing.
- Completion of the prototype UNIX SAGA Pascal Make facility; commencement of testing.
- Completion of an experimental Proof Management System.
- Implementation of the Screen Editing Facilities of the Editor.
- Implementation of an RRP Parser Interface for the Editor.
- Implementation of SAGA Version Control using RCS.
- Berkeley Pascal, Ada, C, Prolog Editors.
- Experimental SAGA pretty printing facilities.
- Experimental Program Transformers.
- Design of the SAGA Attribute Evaluation Scheme.
- Design of an incremental compilation facility.

2. Project Overview

The SAGA system is a software environment that is designed to support most of the software development activities that occur in a software lifecycle. The system can be configured to support specific software development applications using given programming languages, tools, and methodologies. Meta-tools are provided to ease configuration. The SAGA system consists of a small number of software components

that are adapted by the meta-tools into specific tools for use in the software development application. The modules are designed so that the meta-tools can construct an environment which is both integrated and flexible. A description of the SAGA project was published at an ACM Software Engineering Symposium [Campbell and Kirslis, 84] and is included as Appendix A. Several major components of the SAGA system have been completed to prototype form and the method of their construction has been documented.

The project has also been concerned to design SAGA as an evolutionary system and a study of the requirements of such a system was undertaken last Fall with Visiting Professor Peter Lauer. The conclusions of the study are published in the paper RECIPE [Campbell and Lauer, 84], which is included as Appendix B.

A basic SAGA editor is available with both line editing and screen editing modes. We have constructed example editors for Ada, Pascal, Prolog and C. Olorin, the SAGA parser generating system, has been augmented to allow specification of languages by means of Regular Right Part grammars. An incremental parser for the editor has been built that accepts the tables produced for Regular Right Part LALR(1) grammars. This scheme and a system for semantic evaluation of edited languages using attribute grammars is documented in the Ph.D. Preliminary Proposal of George Beshers which is included as Appendix C.

The SAGA Symbol Table Manager is completed to prototype form and is documented in Paul Richard's Masters Thesis which is included in Appendix D. The Symbol Table Manager is now being integrated into the editor and an example Pascal cross-referencing facility is being built. Make, the SAGA separate compilation facility is completed to prototype form and is documented in Wayne Badger's Masters Thesis which is included as Appendix E. A Berkeley Pascal Make has already been integrated into the Pascal SAGA editor. Testing of the prototypes is in progress.

The significant results from this year's research are detailed in the following sections. A tape containing many of the completed prototypes will be sent to NASA in

September for testing and comment.

3. SAGA Editor

During the past half-year, a basic screen interface has been developed for the SAGA editor. The editor now may be run in either screen mode or line mode. Any soft-copy terminal usable with the Maryland Window Library [Torek] which has a standout (or highlighting) mode may be used with the screen-mode editor. The screen interface permits the user to display a screenful of program text from the file being edited. The editing cursor is represented by the terminal's cursor, which is moved over the displayed test and used to select tokens, lines, or trees for editing operations. Intra-token character editing will be added in the near future. New text may be inserted simply by typing it at the cursor's position. The text is tokenized line by line, and parsed at the end of the insertion. Commands are invoked by single-touch control characters, (including function keys if present on the terminal and enabled in the editor), and by a line-mode command escape to execute line mode and user-defined editor commands. A more complete discussion of the editor appears in [Campbell and Kirslis, 84], included as Appendix A.

A new feature was added to the editor to allow separate processes to be invoked to analyze or manipulate the parse tree during an editor session. The *filter* command takes the name of a program to be run, and an optional sub-tree specification to be passed to the program. When this command is invoked, the editor writes its internal data to disk, waits while the program runs, and then reads back in, possibly reparsing, sections of the tree specified by this program. When combined with the user-defined command facility, the filter command allows easy invokation of sophisticated operations on the parse tree. The program which is invoked accesses the tree through an interface which provides function calls to read and write fields in the parse tree. This facility has been used to write programs to pretty-print the parse tree and to transform programming statements from one form to another; for example, to transform a 'for statement' into a 'while statement'. We hope to use the scheme to

connect the editor to an *expert* system that helps a user seeking guidance in a program development or debugging task.

Two parser-generating systems can presently be used to provide a language specification for a SAGA editor: Mystro and Olorin. In addition, any parser-generating system which can meet the editor's LALR(1) style incremental parsing interface can also be used.

3.1. An Editor for Ada

Work is continuing on development of a language-based editor for Ada, using the editor generation facilities of the SAGA environment. The purpose here is not only to provide a rigorous test of the facilities on a large, complex source language, but to produce valuable tools for Ada software creation. The Department of Defense's Ada initiative guarantees the need for such tools on future installations of SAGA, and our effort aims to prepare for this eventuality.

As of this report two editors exist: one based on the Olorin translator writing system, being developed by George Beshers for the SAGA project, and one based on the Mystro translator writing system. Each of these editors accepts the full Ada language as given in the Ada Reference Manual [ARM, 83], and each is complete through the lexical analysis and parsing stages of translation. Current efforts include incorporating the symbol table manager [Richards, 84] into the editor, with a tentative completion date of late August, 1984. Initial validation of the editor, concurrent with the testing and refinement, will be continuing through the fall of 1984.

Accompanying this work is an evaluation and comparison of the two translator writing systems. Initial indications are that the Olorin system provides a faster specification time than the Mystro system. This results from Olorin's use of regular expressions/extended-BNF as a specification language, versus Mystro's use of standard BNF. For Ada this is particularly helpful as the Ada Reference Manual uses extended-BNF in its specification of the language syntax. In general, if the

implementor is familiar with either regular expressions or extended-BNF, completion of the lexing/parser phases of the editor is reduced to a matter of days. Further, the input grammar to Olorin can be easily adapted from a syntax chart of the language, with changes necessary only to make the grammar LALR(1). This property of the system is an obvious plus when one considers that syntax chart specifications of programming languages are becoming much more prevalent than the traditional Backus-Naur specifications. For Ada we used a syntax chart produced by Frank DeRemer, Tom Penello and W.M. McKeeman using an automated translator writing system at the University of California at Santa Cruz.

The editor produced by Mystro is at the same level of completion as the one produced by Olorin, though its implementation took considerably longer. The lack of an OTHERWISE clause in Berkeley Pascal incapacitated the lexical analysis phase of Mystro, requiring the hand-coding of an analyzer similar to one that was created for the Pascal editor. In addition, care was needed to ensure that the input grammar remained LALR(1), as even small modifications to the source grammar [Wetherell, 81] tended to make it non-LALR (1) when fed into Mystro. The difficulty seemed to stem from the inability of Mystro to choose between two default reductions in the same state, even given disjoint follow sets. (A more complete discussion of problems with Mystro is contained elsewhere in this report.) In its current incarnation, however, Mystro is more versatile than Olorin as it allows the user to invoke more options for the system output, including cross-reference tables, extensive error listings and so forth. These and other factors are being considered in the ongoing evaluation of the two translator writing systems. A decision regarding which of the two to make the primary driver for the editor generating system should be made by late 1984.

Still to consider in the full implementation of an Ada editor are the semantic analysis and code generation stages of translation. A projected back-end for the editor would include these operations, with the use of the DIANA intermediate code for Ada as the object language currently being explored. Linkage and referencing errors may need to be detected by a module separate from the editor, thus requiring further

investigation of Ada and its operating environment. These facets of the project will continue to be attended through the end of 1984.

For a listing of the Ada input grammars to Mystro and Olorin, see Appendices F and G at the end of this report.

3.2. SAGA Pascal Editor: Intermediate-Code Generation

A Pascal code-generating utility is under development which will generate intermediate code directly from the parse tree produced by the SAGA Pascal editor; when the Pascal source is re-edited, the utility will modify that intermediate code to reflect the changes made.

The first goal is a code generator. Its input will be: 1) the parse tree file from the Pascal syntax-directed editor, and 2) symbol table information from the SAGA Symbol Table Manager. Its output will be the "c-code" expected by the *f1* pass of the Unix FORTRAN 77 compiler; *f1* is used by the Berkeley Unix Pascal compiler as its second pass.

The second goal is an incremental recompilation facility. Its input will be: 1) the input and output from part one, along with 2) information about the modifications in the edited program text, from the SAGA Pascal Make facility. Its output will be the c-code file produced by part one, suitably altered so that this is now a translation of the edited text, rather than the original.

3.3. An Editor for Prolog

A language-oriented editor for Prolog has been developed using the SAGA editor generation tools. Prolog is a logic based programming language. Appendix H contains an introduction to Prolog. The editor was easy to construct using the SAGA tools. A pre-existing grammar in BNF was put into a format suitable for input to the Mystro parser generator. The lexical specification was modified, and the editor was constructed. Construction of an editor for Prolog was an interesting exercise that demonstrated the utility of the SAGA tools, and produced an editor which may be useful in

future development efforts. Because Prolog is a very high-level language, it has been promoted as a fast prototyping tool.

4. Mystro Parser-Generator

We are presently converting to version 7.0 of Mystro, which provides some of the support for LALR(1) grammars which we require. We converted a PRIME computer version of Mystro to run under Berkeley UNIX versions 4.1 and 4.2, and added some code needed to generate additional tables needed in our environment, in particular, a table of non-terminal token names, which aids editor debugging and grammar preparation.

The present version of Mystro provides inadequate support at the lexical analysis level. Although a case-statement fragment is generated for certain lexical categories, the lack of an *otherwise* clause in our version of Pascal makes its use impossible, and we have written our own code to directly query the terminal token tables instead. A larger problem is the difficulty of specifying generic token classes for unusual token classes. A typical situation is the specification of a grammar to specify Mystro grammars. Since we have a language-oriented editor, it is desirable to use it when preparing grammars for other languages. Yet there is no way to describe a non-terminal class except with an angle-bracketed terminal and a manually written code fragment to recognize that class. The fact that a manually written fragment is necessary precludes automatic generation of editors with Mystro for many languages. A useful extension to the Mystro system would be an additional section in the input grammar file in which a regular expression could be used to specify the construction of a terminal class that is referred to by an angle-bracketed terminal. A standardized piece of code could then either be generated from these expressions, or provided to pattern match input characters against this specification during the lexical analysis phase, greatly increasing Mystro's applicability.

An additional difficulty encountered with Mystro reduce-reduce conflicts during the generation of an Ada editor is described in the Ada editor section of this report.

5. Olorin and Regular Right Part Grammars

Since the 1983 year end report, a new version of the Olorin system has been developed. The new system permits the editor builder to specify his language using *regular right part grammars.* Regular right part grammars are context free grammars, except that the right hand side of all productions can be any regular language, not just a simple string. This means that the syntax charts used by many authors, including N. Wirth, can be translated directly into a SAGA editor (assuming the syntax charts are not too ambiguous). Thus only the non-terminals appearing in the language definition need to be present in the editor. The result of this work is that the parse trees are more amenable to structured editing commands, and that the size of the parse tree is reduced by as much as a third in many examples. The new Olorin system has been used to successfully generate editors for Pascal, ADA, and a Cliff Jones grammar. The new Olorin will be included in the September tape.

Appendix C on the Olorin system also discusses the current efforts to add a symbol table and cross reference facility to the Olorin Pascal editor. The principal goal of this research is to develop a language description which can be mechanically translated into efficient editors. The context sensitive portion of this description is based on attribute grammars, and includes *incrementally updatable attributes.* The updatable attributes introduce a carefully controlled concept of state into the attribute grammar. The state information is structured explicitly for efficient processing of symbol table information in an incremental environment. These efforts are in the early development phases, and should be ready by the year end report.

6. Symbol Table Manager

The first version of the SAGA prototype symbol table manager has been completed, and is described in detail in Appendix D. The symbol table manager is the module responsible for storing and retrieving names used in program code, specifications, and other components manipulated by the SAGA environment tools. It provides the data structures and primitive operations to locate names given scoping

rules for a given programming language, store new names, and store and retrieve basic attributes attached to the names. A set of basic primitives to store and retrieve strings is also provided with the symbol table manager.

The symbol table is intended to be integrated into the SAGA editor so it may be used in the attribute grammar evaluator described in the section on Software Specification. As an intermediate step in this integration, an independent utility to generate a cross-reference index for Pascal programs is being constructed, which uses the prototype symbol table manager. This cross-reference generator will be aware of Pascal scoping and typing constructs and will annotate the program correctly when several definitions for an identifier are present. The attributes and data structures used in the cross-reference utility to analyze Pascal typing will form the basis for the integrating the symbol table into the SAGA Pascal editor. The first version of this utility is about fifty percent complete at the present time.

7. Diff/Undo Facility

A *diff* command, in a very basic form, has been added to the editor. The user can view differences between an old version and the current version of the program being edited, and *undo* the differences. The *undo* function does not depend upon the order in which changes were made to the old version.

An existing version control system (RCS) has been adapted to handle SAGA editor files. The version control system can keep several versions of a program readily available using less disk space since it keeps only one complete version and backward differences between versions to recreate any other. The system also maintains some documentation of changes such as the person who made a change, when the change was made, and a short description of the change from the person who made it. These uses of differences with SAGA editor files are explained more fully in Appendix I.

8. Software Specification

Plans are being made for an integrated system, i.e. a set of tools and methods, to support software specification and development using abstract specifications. The system will be based on the Vienna Development Method (VDM)[Jones, 80], which allows the developer to start with a completely abstract specification then refine it through a number of steps into a program. The abstract specifications are based on predicate logic and predefined mathematical data types. The VDM has been used successfully on large software projects, and is suggested as a good choice for automation [Shaw et. al., 84].

Each step in the refinement process and each design decision produces an abstract program [Parnas, 77] from a specification. An abstract program may be described in two ways. An abstract program is: an abstract specification with some design decisions made; and a program in the base language (the language in which the completed program is written) with some modules only specified abstractly. Through a sequence of refinements, the abstract program is transformed into a concrete program in the base language. Each design decision will be supported by some documentation, and a rigorous argument, or proof, that the design decision produced an abstract program that matches the original specification.

The system will support versioning of abstract programs and the supporting documentation and proofs. The idea of versions and alternatives has been used in systems for design of VLSI circuits [Katz and Lehman 82]. A design begins as a initial version. During the development, a new in-progress version can be created from an initial version. A version may be split into several alternatives which may be developed independently. One of several alternatives may be selected to form a new version. At any point during the development process the developer can change to a previous or alternative version.

9. Proof Management

To aid in the development of formal proofs, such as those arising in formal program verification, a proof management system is a desirable tool. A proof management system can make large, complex proofs easier to write, modify, and understand. More importantly, a proof management system also provides a means for the validity of a proof to be checked automatically.

A prototype proof management system has been implemented. The system is based on a structure editor for trees. In this system a proof is represented by a tree, with each node in the tree representing a formula derived from that node's children. In the proof trees the validity of a node depends only on that node's relationship with its immediate children.

By itself the editor has no ability to certify (check) inferences. However, through the editor's *call* command, external programs can be invoked to examine and alter the proof tree being edited. A number of such external programs have been written and can be called from the tree editor. These include interfaces to a resolution based theorem prover, interfaces to a theorem prover based on term rewriting systems, a program to include previously proven theorems as lemmas for the proof being edited, and programs to display proofs.

Currently the interfaces to theorem provers rely on an awkward, prefix notation for first order formulas. Although the syntax is rather unpleasant, the notation is sufficient and was readily available. It is anticipated that a better syntax will be implemented in the future. A more complete description of the editor and associated programs is given in appendix J.

10. Summary

We believe the SAGA project has made significant progress in this last half-year. Several components have developed to the point where tools can be built that use them. The Olorin parser generator has developed to the point where it offers a useful

alternative to the Mystro parser generator: allowing syntax diagrams to be easily coded for use in editor production. The viability of the SAGA editor recognizer approach is no longer in question and the major performance issues can now be addressed. Incremental reparsing provides an efficient way of maintaining data modified by the editor. In particular, we have shown that the editor user interface is flexible and allows arbitrary modifications without compromising the editor's ability to detect invalid programs.

Several of the tools (the editor, symbol table, diffundo, and RCS) are being integrated, with encouraging results. The Ada editor is a significant milestone in the SAGA effort as it demonstrates the capability to support a complex programming language. The Make system and the diffundo system are both tools that benefit from the modular interfacing of several SAGA components. Application of the symbol table manager to support a Pascal cross-reference tool is making progress. It is interesting to note that it has been possible to construct several simple tools like a pretty printer and program transformer out of the existing tools quickly and with little effort.

By adopting Clifford Jones' rigorous design methodology, we believe that we can demonstrate how the specification, design, and verification stages of software development can be integrated with the programming, version control, and testing stages of the lifecycle. The proof management system has made good progress and has been used to manage several simple, but lengthy proofs. Although it is in experimental form, the proof management system now allows us to proceed to associate the design process with an automated mechanism for verifying that design process.

To conclude, we have little doubt that the tools we have produced in the last six months will make the next six months a very exciting period as the simple components we have designed are integrated together to form a powerful software development prototype environment.

References

[ARM, 83] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, U.S. Dept. of Defense, Springer-Verlag, 1983.

[Campbell and Kirslis, 84] Campbell, Roy H., and Peter A. Kirslis, "The SAGA Project: A System for Software Development," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA., Apr., 1984.

[Campbell and Lauer, 84] Campbell, Roy H., and Peter E. Lauer, "RECIPE: Requirements for an Evolutionary Computer-based Information Processing Environment," Presented at the Software Process Workshop sponsored by ACM, BCS, ERO, IEE, IEEE Computer Society, Surrey, Feb. 6-8, 1984.

[Jones, 80] Jones, Clifford B., *Software Development: A Rigorous Approach*, Prentice Hall International Series in Computer Science, 1980.

[Katz and Lehman, 82] Katz, R. H. and T. Lehman, "Storage Structures for Supporting Versions and Alternatives," Computer Sciences Tech. Report #479, University of Wisconsin-Madison, July 1982.

[Parnas, 77] Parnas, David L., "The Use of Precise Specifications in the Development of Software," Proc. IFIP Congress, 1977, pp. 861-867.

[Richards, 84] Richards, Paul, *A Prototype Symbol Table Manager for the SAGA Environment*, Master's Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1984.

[Shaw et. al., 84] Shaw, R.C., Hudson, P.N., and N.W. Davis, "Introduction of a Formal Technique Into a Software Development Environment (Early Observations), " Software Engineering Notes, Vol 9, No 2, April 1984, pp. 54-79.

[Torek] Torek, C., *The Maryland Window Library*, Dept. Computer Science, University of Maryland, College Park., MD., 20742. No date given.

[Wetherell, 81] Wetherell, C.S., "Problems with the Ada Reference Grammar," ACM SIGPLAN Notices, Vol. 16, no. 9, September 1981, pp. 90-104.